ESCI 7205 – Data Analysis in Geophysics - Homework #2 – 9/10/13 – DUE 9/17/13.

Examination of Newton’s method for complex valued functions.

Generalize the Newton’s method exercise we did in lab #5 (9/10/13) to use complex numbers.

This will allow you to find all three roots of the cubic equation – Z3-1=0, where Z is now a complex number. The example in class, using only real values for x, produces only the real root.

We will actually be doing a 2-d version of the example for which I showed results in class. In the example we were interested in the convergence rate as a function of starting value, rather than just the application of Newton’s method for a single starting value as we did in the exercise.



Since Matlab handles matrices as numbers, the basic program structure is the same - we just have to substitute the name of a matrix into the code from class.

The complication will be that we don’t stop the loop when we have converged or diverged. We will have to use some way to save who has converged and who has diverged in logical matrices. We will also have to save the iteration counter for convergence in a matrix (this will be the result we plot at the end).

To make life easier we will also just save the root we found during the calculation and sort out which one it is for coloring the plot later.

Soooo-

You need to make a complex matrix whose elements are the starting values on an x-y grid. Let the grid range from -2 to 2 in both X and Y directions.

The easiest way to do this is to start with an integer grid.

Start making the grid by creating a 1-d vector that is 2N+1 long using [-N:N]. This will map the grid points directly into the (x,y) points later on.

So we will make a grid like that shown below. The circle is the origin. The point labeled (x,y) is at (1,1). The complex plane maps into the (x,y) plane through the relation between x, y and z given.



To make the grid we need matrices of all the x and y values and then add them together. Let xi, yi and zi be the integer x, y and z matrices.

$$xi=\left(\begin{matrix}-N&-\left(N-1\right)&…\\&\vdots &\end{matrix}\begin{matrix}N-1&N\\&\end{matrix}\right)$$

$$yi=xi'$$

and we combine these matrices to make our matrix of complex numbers

$$zi=xi+i\*yi$$

When we need the actual z values of the grid points on our grid from -2 to 2 we just rescale

$$z=zi\*2/N$$

and we will use z in the calculations.

Now we can iterate (loop) to find the roots for each starting value (every point in the matrix z).

What we will have to change from the example. We are now using matrices so we need to use the element-by-element operators

.^ and ./

when we calculate f, the slope, and the adjustment to z.

We will still have to test for convergence after each iteration, but now we will have to test the matrices and figure out which elements have converged and then save that information and also remove those values from further calculation. There are a lot of ways to do this. I’m going to suggest one that is easy conceptually, but is probably not the most efficient.

So how to find the elements that meet some condition. Use the Matlab function “find”. For example (roots is a predefined matlab function so that nice variable name is not available)

rootz=find(abs(f(:))<epsilon);

This will look at f as a vector and return the indices of elements in the vector that meet the condition (f is really a 2-d matrix and one could get the results in terms of the row and column position, but it is easier to use the 1-d representation.).

We can now use the indices listed in rootz to save the appropriate values (the value of the root we found, removing that z from further calculations, etc.)

We do the same thing looking for zeros of the derivative, and remove z values that diverge from the calculations and flag them as non-converging.

The easiest way to keep track of all this ancillary information is with a bunch of other matrices.

Make a matrix of NaNs for the convergence count, the value of the root, and values of z that are being removed from further calculations (because they converged or are diverging).

(Why using NaNs? When one does any kind of arithmetic with a NaN the result is a NaN. When doing things like plotting, the NaNs are [usually] ignored. So setting something equal to NaN effectively removes it from the calculation, plot, etc. Matlab is smart enough [I think] to stop calculating when it sees a NaN and just puts a NaN into the answer.)

As we find roots, we will fill in the appropriate elements (using the output of the find function) with the convergence counter, the value of the root, etc. We will also set the values that have converged or will diverge to NaN in the matrix for z so they do not get included in the following loops.

Once we have iterated the maximum number of times, we have to figure out the root associations. For now you will see something that looks like this using the matlab function imagesc. (the first figure above, from the example in class, is actually a plot of the function below along the real axis).



To make life simpler, we will use the fact that we already know the 3 roots, so we can simply test our estimations of the root against the three known roots to identify which root we converged to. (If not we would need to find the set of roots represented by the not all exactly equal estimations we have).

Use the following to calculate the roots (where did this come from?).

rts=exp(i\*2\*pi\*[0 120 240]/360);

We initially just saved the roots and now have to figure out how to separate them. Once we have done that we can build a matrix with red, green and blue associated with the root to which each z converges and we will get something that looks like this.

One can play with the values to make the color gradients more visible, etc.



Pseudo code – something that looks like a program, but is not in any specific programming language. Maps into program pretty directly – almost line for line.

0) [count like computer to fix my mistake of initially not counting this step without having to renumber everything!] Clean up (things like - clear, clf, etc.)

1) Set the size of the grid using numbers related to your working space, not computer language array indexing rules. This makes it easier to figure out what you are doing and debug (the errors that cannot exist in any program you wrote).

2) Make the integer vector of grid points in your indexing scheme.

3) Find out how long it is and save this value

4) Make the array of XI values (I’m calling this XI to indicate that it has my grid counting integer values, not the actual x values to be used in the final calculation). This array can be made from your vector of grid, using it as a ROW vector, and then filling out the square matrix by repeating it into each row of the array you are building.

5) Make the array of YI values. The easy way to do this is to transpose the vector for XI.

6) Now make ZI=XI+i\*YI

7) Now set up some metadata arrays –

8) We need an array to hold the number of iterations to convergence. It will be the same size as XI, YI and ZI. Initialize it with NaN’s. When we are done the elements of this array will have the number of iterations for the starting z for that position, or NaN if it did not converge.

9) We need an array to hold the value of the root we found. Initialize it with NaN’s for the same reason as above.

10) Now make array, z, with the complex values we will use as the starting points for the iteration. This is easy – just multiply the matrix ZI by the limits of the region you are searching (-2 to 2 in this case, so multiply by 2) and divide by the number of points you specified as the grid size. So if the number used in 1 was 10, that means that your grid goes from -10 to 10, for 21 points in each direction. (You multiply ZI by 2/10)

11) Set the maximum number of iterations

12) Do loop from 1 to the maximum number of iterations (could also check to see if have completed the calculation – i.e. every starting value has converged or diverged – and stop – will not do this yet.)

 13) Calculate function, f, for whole grid

 14) Check convergence of all elements in grid by getting list of elements meeting convergence criteria.

 15) If any elements have converged -

 16) Save count of iterations to convergence in appropriate array.

 17) Save the value of the root (is the value of z used to calculate f) in appropriate array.

 18) Set appropriate z value to NaN to stop further calculations.

 19) Calculate slopes (derivative)

 20) Check for zero (actually very small) slopes/derivatives grid by getting list of elements meeting “zero” criteria.

 21) Set zero slopes to NaN (stops calculating).

 22) Set appropriate z value to NaN to stop further calculations.

 23) Calculate new z=z-f/slp.

24) End of loop.

Have answer – now have to do graphics.

Can plot it directly without root identification (tells us how fast converges, but not which root).

25) To do “pretty” make vector with actual x and y values along axis (instead of getting integer point number counter). Uses same scaling as to make z matrix, but now just do to the vector of integer N counters.

Now we have to get our data (number of iterations to convergence – smaller is better) into something the plot routine “imagesc” likes. imagesc takes values from 0 to 1 and changes them into colors. If we give it a NxN matrix it will map the single values at each point into a color. Prepare the data for plot.

26) Find the maximum number of interations

27) Call imagesc with “imagesc(axisvalues, axisvalues, iterationsmatrix/maxiterations)”, where axisvalues is the vector with the actual values between -2 and 2 for each grid point in the x and y directions, and interationsmatrix is the array where you have the number of iterations (or NaN) divided by the max number of iterations to make the values go from 0 to 1. (you could also use

(maxiterations-interationsmatrix)/maxiterations

to make the faster ones “brighter” – this will be needed in the next step where we id the roots).

28) Now make a 3-d matrix where each level has the convergence speed for one of the three roots. We will map this data into red, green and blue.

29) Again, due to the limitations of Matlabs indexing capabilities and abilities to handle >2-d matrices easily, we will have to use linear indexing into the 3-d matrix. To do this we will need to know the number of elements in each “layer” of the array. Calculate this. It is the product of the dimensions of the matrix.

30) Loop over the three roots

 31) Find the indices for root being examined

 32) Store in 3-d matrix

33) End of loop

34) Do same scaling as before, but now want faster convergence (smaller numbers) to have larger values (1 is color full bright, 0 is color “off” – so have to swap values around – see 27).

35) Plot it!